Poznan University of Technology

End of term project assignment

Chinese Operating System

Based on Linux architecture

Written in C++

Authors:
Krzysztof Bojakowski
Filip Fischer
Tomasz Lewandowski
Michał Pietrzak
Mateusz Szynka

Supervisor:
dr Jerzy Bartoszek

March 2015

Contents

Preface **4**

1 Kernel **5**
1 General Overview . . . 5
1.1 Description . . . 5
1.2 An entry point . . . 5
1.3 Kernel's Loop . . . 5
2 Classes . . . 5
2.1 Object . . . 5
2.2 Module . . . 6
2.3 Kernel . . . 7
3 Usable methods and shell commands . . . 7
3.1 Module . . . 7
3.2 Kernel . . . 8
3.3 Shell commands handlers . . . 8

2 Shell **9**
1 General Overview . . . 9
1.1 Description . . . 9
1.2 How does it work . . . 9
2 The Shell Class . . . 10
2.1 Definition . . . 10
2.2 Usable methods . . . 10

3 Logger **12**
1 General Overview . . . 12
1.1 Description . . . 12
1.2 Abstraction . . . 12
2 Classes . . . 12
2.1 Logger . . . 12
2.2 Channel . . . 13
2.3 Level . . . 14
2.4 Console Logger . . . 14
3 Usable methods . . . 15
3.1 Logger . . . 15
3.2 Channel . . . 15
3.3 Level . . . 15

3.4 Console Logger 15

Modules 17

1 Processor 17
1 Classes overview 17
1.1 ExecutiveUnit, Processor, Scheduler, Interpreter 17
2 Data structures 18
2.1 Processor 18
2.2 ExecutiveUnit 18
2.3 Scheduler 18
2.4 Interpreter 19
3 Usable methods 19
3.1 Processor methods 19
3.2 Scheduler methods 21
3.3 InterpreterHandler methods 22
3.4 ExecutiveUnit methods 22
4 Shell - step by step operations 22
4.1 Shell commands 22
5 Interface 23

2 Memory by Filip Fischer - The part is not included and will be given at another time 23

3 Processes 24
1 Data Structures 24
1.1 Process structure 24
2 Code to perform the algorithm 25
2.1 Processes methods and other ingredients 25
3 Interface to present data changes 27
3.1 Commands for Shell 27
4 Step by Step 27
4.1 Shell 27
5 Interface to cooperate with other modules 28

4 Files 29
1 Action concept 29
1.1 Filesystem 29
2 Data structures 30
2.1 Filesystem 30
2.2 File 30
2.3 Block 31
2.4 Inode 31
2.5 Data 32
3 Usable methods 32
3.1 Filesystem 32
3.2 File 34
3.3 Serialization module 34

5 Users **35**
1 General Overview . . . 35
1.1 Description . . . 35
1.2 Type definitions . . . 35
1.3 Built-in constants . . . 36
2 Data structures . . . 36
2.1 User . . . 36
2.2 Group . . . 37
2.3 Permissions . . . 37
3 Classes . . . 38
3.1 Module . . . 38
3.2 Users Manager . . . 38
3.3 Groups Manager . . . 39
3.4 Permissions Utilities . . . 40
3.5 Permissions Manager . . . 40
3.6 Priority Manager . . . 40
3.7 Encryptors . . . 41
4 Usable methods . . . 42
4.1 Module . . . 42
4.2 Users Manager . . . 42
4.3 Groups Manager . . . 42
4.4 PermissionsManager . . . 43
4.5 PriorityManager . . . 43
4.6 Encryptors . . . 43
5 Shell commannds handlers . . . 44
5.1 Users managment commands . . . 44
5.2 Groups managment commands . . . 44
5.3 Permissions managment commands . . . 45
5.4 Priority managment commands . . . 45

Project summary **46**

Preface

Purpose This project was created as a result of end-of-term assignment in the laboratory part of the subject of Operating Systems - Design and Architecture. The main purpose was to create an operating system based on Linux operating system architecture using C++ programing language.

Design Design part of the project was started with joint agreement on coding style (based on Google C++ Style Guide) which was then written by Tomasz Lewandowski as a *"Coding style"* document attached as Appendix 1. The unity of the code created a possibility to read each others code with ease and created more efficient ways of work.

Additional libraries The complexity of an assignment has created a demand for additional tools and libraries to help in the developing process. As a team we were using Boost library features such as `boost::smart_ptr`, `boost::weak_ptr` to maintain memory leakage issues in the merging and deploying part of the development process. Some of us also used `boost::bind` for binding commands for the shell and `boost::regex` for complex data processing and management.

Chapter 1

Kernel

by Tomasz Lewandowski

---

1 General Overview

1.1 Description

The general idea behind the creation of the Kernel, was to make an entry point in the system. There was also a need to provide easy communnication between the Modules, and that is the main role of the Kernel. The goal was also to provide easiness during the merging process.

1.2 An entry point

By providing an entry point the main function has only 3 lines. Creating the Kernel class, initializing it, and running it's main loop. The Kernel initializes each of the Modules automatically during kernel's initialization process. This approach provides a separate entry point for every module in the system, and guarantee clean initialization process. The second advantage of this method is the ability to catch every exception that is thrown during initialization process. The third one is the ease of development further modules.

1.3 Kernel's Loop

Kernel works in the loop until the shutdown procedure is called. To provide step-by-step working functionality, each iteration Kernel is calling a Shell step and blocks on it until the user type in the shell command and press the enter button.

2 Classes

2.1 Object

Simple object class. Many classes in the project inherits from that class. It was designed to provide Java-like toString() functionality.

```
class Object
{
  public:
    Object();
```

```
    virtual ~Object();

    /*
      Returns class name with leading namespaces.
      E.g. sop::Object

      Every class should override this function.
    */
    virtual std::string getClassName() const;

    /*
      Returns class name and value of pointer to this object.
      May be overriden if necessary.
    */
    virtual std::string toString() const;

  protected:

  private:

};
```

2.2 Module

Module is an abstract class. Every module inherits from that class to provide access to module specific methods.

```
class Module :
  public sop::Object,
  public sop::Initializable
{
  public:
    /*
      Constructor for Module element.
      Params:
          kernel - pointer to kernel that module is created for.
          Pointer should be valid for whole module lifetime.
    */
    explicit Module(sop::system::Kernel *kernel);

    virtual ~Module();

  protected:
    /*
      Pointer to kernel, that module is contained in.
    */
    sop::system::Kernel *_kernel;

  private:

};
```

2.3 Kernel

Kernel class. It is the heart of the system. Kernel consists of different modules (e.g. processor, file system). It manages and initializes modules, shell and the logger. Every module, through the Kernel, can access shell, logger an the other modules.

```
class Kernel :
  public sop::Object,
  public sop::Initializable
{
  public:
    enum State{
      ERROR,
      PRE_INIT,
      INITIALIZED,
      RUNNING,
      SHUTTING_DOWN
    };

    /*
      Creates kernel with default logging level.
    */
    Kernel();

    /*
      Creates kernel with given logging level.
    */
    explicit Kernel(uint16_t logging_level);
    virtual ~Kernel();

  protected:

  private:

    mutable State _kernel_state;
    boost::shared_ptr<sop::logger::Logger> _logger;
    sop::system::Shell _shell;
    sop::users::Module _users_module;
    sop::files::Module _files_module;
    sop::processes::Module _processes_module;
    sop::memory::Module _memory_module;
    sop::processor::Module _processor_module;
    std::vector<sop::system::Module*> _modules;
};
```

3 Usable methods and shell commands

3.1 Module

- `virtual void initialize()= 0;` - Should be defined in inherited classes and should initialize everything that module needs to work (e.g. register functions to shell).
- `sop::system::Kernel * getKernel();` - Returns the pointer to the Kernel that the module was created

for.

3.2 Kernel

- `virtual void initialize();` - Initializes all the modules and handles all the exceptions that may occur during this process.
- `void run();` - runs the Kernel's loop
- `sop::logger::Logger * getLogger();` - returns pointer to the logger
- `sop::system::Shell * getShell();` - returns pointer to the shell
- `sop::users::Module * getUsersModule();` - returns pointer to the users module
- `sop::files::Module * getFilesModule();` - return pointer tos the files module
- `sop::processes::Module * getProcessesModule();` - returns pointer to the processes module
- `sop::memory::Module * getMemoryModule();` - returns pointer to the memory module
- `sop::processor::Module * getProcessorModule();` - returns pointer to the processor module
- `std::vector<sop::system::Module*> getModules();` - returns the vector of pointers to all modules
- `bool isShuttingDown()const;` - returns true if system is currently shutting down or false otherwise
- `void shutdown()const;` - shuts down the system

3.3 Shell commands handlers

- `void helpHandler(const std::vector<const std::string> & params);` - command handler for the help command that prints all the commands registered in the shell.
- `void shutdownHandler(const std::vector<const std::string> & params);` - command handler for the shutdown command that closes the system.

Chapter 2

Shell

by Tomasz Lewandowski

---

1 General Overview

1.1 Description

The idea behind creating a shell, was to create single and consistent interaction with the user, as well as forcing the system to work step by step. The desing of the class provides easy to use methods to register shell commands, and functions or methods that will handle them. It eliminated the problems with reading from the default input stream. The next advantage of this approach is that the modules using this API can stay separate from the other modules. Every team member could create and register his own commands to test his module functionality. Since the only interaction with the keyboard was done by the shell and processes, there would be no issues during the merging process as long as the shell commands would have different names.

1.2 How does it work

The goal was to connect a command name with the simple function or the method of a specific Object. The second requirement was to allow every developer to dynamically register his own commands therefore Object could be of any class. This eliminated the possibility of using a raw funtion pointers, and forced to design this mechanism in a more sophisticated way.

The design is based on the Boost Signals2 library. The shell contains a dictionary that maps the command's names, which are a strings, to the CommandHandlerSignals. Each CommandHandlerSignal has a CommandHandler or an object's method binded with a specific object, so when the signal is invoked, then the proper handler is invoked as well.

One action of the shell is called "a step". Every step the shell reads one line of the input stream, parse it and divide it into arguments. The arguments are a words separated by a space or the phrases written in the quotation marks. To write quotation marks in the arguments an escape signs are provided. The command name itself is also treated as an argument, and it is the first one. If there was a problem with the input line then the shell ends it's step.

After parsing, the shell is looking for the command name in the map and if it exists, invokes the CommandHandlerSignal that the name is mapped to. The signal is invoked and all the arguments (including the command name) are passed to the command handler, so it can do whatever the programmer planned it to do. After invoking the handler, shell ends it's step.

2 The Shell Class

2.1 Definition

```
namespace sop
{
  namespace system
  {
    class Shell : public sop::Object
    {
      public:
        /*
          Command handler function signature.
        */
        typedef void (CommandHandler)(const std::vector<const std::string>&);

        /*
          Command handler function signal signature.
        */
        typedef boost::signals2::signal<CommandHandler> CommandHandlerSignal;

        explicit Shell(sop::system::Kernel * kernel);
        virtual ~Shell();

      protected:
      private:

        /*
          Kernel, that shell is runned on.
        */
        sop::system::Kernel * _kernel;

        std::string _last_input_line;

        /*
          Maps commands names to handling functions/methods.
        */
        std::map<std::string, boost::shared_ptr<CommandHandlerSignal>> _commands;
    };
  }
}
```

2.2 Usable methods

- `void step();` - performs one shell step
- `std::vector<std::string> parse(const std::string & line);` - parses the given line and returns vector of arguments as a result or an empty vector if there was any problem during the process
- `static bool hasParam(const std::vector<const std::string> & param, const std::string & param_name);` - utility function for handler's developpers. It returns true if the given vector contains the parameter specified by the name or false otherwise.
- `static std::string getParamValue(const std::vector<const std::string> & param, const`

`std::string & param_name);` - another utility function for handler's developpers. Parameters can exist in form of *name value* (e.g. *-p mypassword*). This function returns the value of the parameter specified by the name, or zero if the parameter was not found.

- `bool registerCommand(const std::string & command, CommandHandler * function);` - registers a command specified by command and maps it to the function specified by function. Returns true on success or false otherwise.
- `template <class ClassType> bool registerCommand(const std::string & command,void (ClassType::*method)(const std::vector<const std::string>&), ClassType * object);` - registers a command specified by a command. It maps the command to the method of the object specified by object. Example invokation of the method: `registerCommand("useradd",&Module::cH_useradd,&module);` where the module is of class Module.

Chapter 3

Logger

by Tomasz Lewandowski

---

1 General Overview

1.1 Description

The motivation to create Logger was to eliminate direct writing to default output stream and to create consistent logging style in every module. Thanks to this every module's output looks exactly the same, allowing easy bug fixing in the project. The code consistency is important as well, so every person in team can have better overall understanding of the code in the whole system. Logger was designed to be easy to use for every team member, so the channel and specialized function for every module was provided. Logger supports different logging levels depending on significance of events therefore it can be set to output all, or these which are important at the time, or even none at all.

1.2 Abstraction

Abstraction was a way to provide ability to log in different destinations i.e. a default output stream, a file, a server etc. In the system there is included the most primitive way of logging - a default output stream.

2 Classes

2.1 Logger

Logger is an abstract class that provides logging functionality. It does not save the logs anywhere so the Logger implementations have to do it on their own.

```
namespace sop
{
  namespace logger
  {
    class Logger : public sop::Object
    {
      public:

        /*
```

```
10          Creates logger which logs only SEVERE logs.
          */
12        Logger();

14        /*
            Creates logger which logs logs with level <= logging_level.
16        */
          explicit Logger(uint16_t logging_level);
18        virtual ~Logger();

20      protected:

22      private:

24        /*
            Logger logs every log with level <= _logging_level.
26        */
          uint16_t _logging_level;
28

30    };
    }
32  }
```

2.2 Channel

Channel class groups the constants of built-in logging channels id's. It also provides functionality to change channel id to it's name.

```
namespace sop
2 {
    namespace logger
4   {
      class Logger : public sop::Object
6     {
        public:
8         static class Channel
          {
10          public:
              static const uint16_t USERS_CHANNEL = 1;
12            static const uint16_t FILES_CHANNEL = 2;
              static const uint16_t PROCESSES_CHANNEL = 3;
14            static const uint16_t MEMORY_CHANNEL = 4;
              static const uint16_t PROCESSOR_CHANNEL = 5;
16            static const uint16_t KERNEL_CHANNEL = 6;
              static const uint16_t SHELL_CHANNEL = 7;
18
            private:
20            Channel();
          };
22
          //...
24    };
```

```
  }
}
```

2.3 Level

Level class groups the constants of built-in logging levels id's. It also provides functionality to change level id to it's name. The levels were based on Java's logging levels.

```
namespace sop
{
  namespace logger
  {
    class Logger : public sop::Object
    {
      public:
        //...

        static class Level
        {
          public:
            static const uint16_t OFF = 0;
            static const uint16_t SEVERE = 1;
            static const uint16_t WARNING = 2;
            static const uint16_t INFO = 3;
            static const uint16_t CONFIG = 4;
            static const uint16_t FINE = 5;
            static const uint16_t FINER = 6;
            static const uint16_t FINEST = 7;

          private:
            Level();
        };

        //...

    };
  }
}
```

2.4 Console Logger

```
namespace sop
{
    namespace logger
    {
        class ConsoleLogger : public Logger
        {
          public:
            ConsoleLogger();
            explicit ConsoleLogger(uint16_t logging_level);
            virtual ~ConsoleLogger();
```

```
12          protected:

14          private:
          };
16    }
}
```

3 Usable methods

3.1 Logger

- `void log(uint16_t channel, uint16_t level, const std::string & message);` - logs a message with given channel and level.
- `uint16_t getLoggingLevel()const;` - returns current logging level.
- `void setLoggingLevel(uint16_t level);` - sets the logging level.
- `virtual void saveLog(uint16_t channel, uint16_t level, const std::string & message)= 0;` - method that outputs the log at given channel with given level. It is invoked by the log method when the log's level is smaller or equal to the loggers' logging level. Method should be overridden by loggers implementations to write the log for example to file or send it through a network.

Besides the above methods logger contains a familly of `void logX(uint16_t level, const std::string & message);` methods, where X is the built-in channel's name.

3.2 Channel

- `static std::string getChannelName(uint16_t channel);` - Returns the string name of the given logging channel. If channel does not have name then the given value is returned.

3.3 Level

- `static std::string getLevelName(uint16_t level);` - returns the string with the name of given logging level. If level does not have name then the given value is returned as a string.

3.4 Console Logger

- `virtual void saveLog(uint16_t channel, uint16_t level, const std::string & message);` - prints the logs on the standard output stream in the form of *CHANNEL:LEVEL:MESSAGE* - each in the separate line.

Modules

Chapter 1

Processor

by Krzysztof Bojakowski

---

1 Classes overview

1.1 ExecutiveUnit, Processor, Scheduler, Interpreter

Processor Processor architecture is based on Intel 8086 microprocessor. I've compressed it a bit, there're 16bit registers (a, b, c, d), instruction pointer, stack, and only two flags (zero flag, sign flag). These're the elementaries to meet the minimum requirements for the linux-similar system implementation.

Scheduler Scheduler is based on linux system algorithms. I've implemented Round robin algorithm mixed with priority algorithm. There're two lists that represents active or unactive processes. The first idea was to keep them as pointers, and whenever new age comes, revert the pointers (that pointer to active array should point unactive array, and pointer to unactive array should point active array). However I've decided to change it. Using bool types - which show me, which of the list is active / unactive I've felt much more convinced what is going on in the system.

Interpreter Commands that interpreter handles are mostly based on machine code. Commands must be 3 chars length, otherwise it won't be interpreted. However the interpreter is included in processor section, what I'll show later. The method used is simple: I ask for chars from memory as long as I don't receive sign of new line, when I do, I split chars on command part, and data part. Then I recognize command part and execute it using the data part.

Data structers Classes are really huge, so below I'll show only data held in them, and later on the rest of these class methods.

Executive Unit The executive unit is just the class created to hold all other classes in it. It puts everything together, and executes it, that's why I called it that way. I needed this class to be able to manage every action of the processor, scheduler or interpreter, and make it work together.

2 Data structures

2.1 Processor

```
namespace sop
{
  namespace processor
  {
    struct processor
    {
      uint16_t a,b,c,d; // 16bit registers of the processor, a , b , c , d
      uint16_t ip; // instruction pointer (counter)
      std::stack <uint16_t> processor_stack; // processor stack
      uint16_t zero_flag; // if (true) then operation result is 0
      uint16_t sign_flag; // if (false) then operation result is positive (>=0)
    };
  }
}
```

2.2 ExecutiveUnit

```
namespace sop
{
    namespace processor
    {
        class ExecutiveUnit
        {
        public:                                    sop::interpreter::InterpreterHandler
             interpreter; // instance of Interpreter
            sop::processor::Scheduler. scheduler; // instance of Scheduler.
            boost::shared_ptr<sop::process::Process> _runningProcess; // actually running process.
            boost::shared_ptr<sop::process::Process> _lastUsedProcess; // last process used.
        private:
            short _quantTimeLeft; // quant time that is left
            short _standardQuantTime; // remembers the standard quant time (initial)
```

2.3 Scheduler

```
namespace sop
{
  namespace processor
  {
    class Scheduler
    {
    public:
        //Methods implemented, will write about it below
    private:
      std::vector <std::queue<boost::shared_ptr<sop::process::Process> >> _first_task_array;
      //list of [active/unactive] processes
      bool _isFirstActive; // tells if the first array is active or unactive
```

```
13      std::vector <std::queue<boost::shared_ptr<sop::process::Process> >> _second_task_array;
        //second list of [active/unactive] processes
15      bool _isSecondActive; // tells if the second array is active or unactive
    }
17  }
}
```

2.4 Interpreter

```
namespace sop
2 {
  namespace processor
4   {
    class InterpreterHandler // will handle bytes received from memory and store program
        information
6   {
    public:
8     //Methods implemented, will write about it below
    private:
10    std::string _command_part; // part fo command (commands, e.g. ADD)
      std::string _data_part; // part for data (registers, values etc.)
12    std::string _program_line; // the one line of program code
    }
14  }
}
```

3 Usable methods

3.1 Processor methods

`namespace sop::processor::ProcessorHandler`

- `static void REBbut16(processor *proc)` - reads one char from input and saves it to register A, command "REU".
- `static void readOneByteFromInputAndSavesItOnYoungestByte(processor *proc)` - reads one char from input and saves it on youngest byte of register A, command "REB".
- `static void printsOutYoungestByte(processor *proc)` - prints youngest byte of register A, command "WRB".
- `static void printsOutYoungestByteAsASCII(processor *proc)` - prints youngest byte of register A as ASCII, command "WRC".
- `static void printsOutRegisterWithoutSign(processor *proc)` - prints uint16 on output, command "WRI".
- `static void printsOutRegisterWithSign(processor *proc)` - prints int16 register on output, command "WRU".
- `static void swapBytes(processor *proc, char reg)` - swapping bytes of selected processor, command "SWB".
- `static void savesReturnAdresOnStack(processor *proc)` - sets IP on stack, and sets offset (from D reg), command "CAL".

- `static void loadsValueFromStackAndJumpOnIt(processor *proc)` - pops value from stack on do "JMP" on it, command "RET".
- `static void saveOnYoungestByte(processor *proc, uint16_t input)` - saves value on the youngest byte of processor A, command "REB".
- `static void multipliesAandB(processor *proc)` - reigster A * register B, and the result is held in C, command "MUL".
- `static void dividesAandB(processor *proc)` - register A / register B, result in C, rest from division is in D, command "DIV".
- `static void doJMP(processor *proc)` - sets offset to given offset value, command "JMP".
- `static void doJIZ(processor *proc)` - sets offset to given offset value if flag zero is active, command "JIZ".
- `static void doJNZ(processor *proc)` - sets offset to given offset value if flag zero is not active, command "JNZ".
- `static void doJIA(processor *proc)` - sets offset to given offset value if first number is greater than second, command "JIA".
- `static void doJAE(processor *proc)` - sets offset to given offset value if first number is greater or equal to second, command "JAE".
- `static void doJIB(processor *proc)` - sets offset to given offset value if first nubmer is lower than second, command "JIB".
- `static void doJBE(processor *proc)` - sets offset to given offset value if first nubmer is lower or equals to second, command "JBE".
- `static void printOutProcessorState(processor *proc)` - prints out all processor fields.
- `static void clearProcessor(processor *proc)` - clears all processor fields.
- `static uint16_t softCharRegisterHandler(processor *proc, char processor_register)` - gets processor register value.
- `static uint16_t* charRegisterHandler(processor *proc, char processor_register)` - convert char into variable and sets a pointer on register.
- `static void registerIncrement(processor *proc, char processor_register)` incrementing register field by one, command "INC".
- `static void registerDecrement(processor *proc, char processor_register)` - decrementing register field by one, command "DEC".
- `static void registerIncrementByValue(processor *proc, char processor_register, uint16_t value)` - incrementing register field by value, command "ADV".
- `static void registerDecrementByValue(processor *proc, char processor_register, uint16_t value)` - decrementing register field by value, command "SUV".
- `static void setRegisterField(processor *proc, char processor_register, uint16_t value)` - setting value into register field, command "MOV".
- `static void compareRegisters(processor *proc, char register_one, char register_two)` - comparing two registers, setting proper flags, command "CMP".
- `static void copySourceRegisterToDestinationRegister(processor *proc, char source_processor_register, char destination_processor_register)` - source means processor register to copy from, and destination means processor register to be pasted in, command "MOR".

- `static void addSourceRegisterToDestinationRegister(processor *proc, char source_processor_register, char destination_processor_register)` - e.g. destination_processor_register = destination_processor_register + source_processor_register, command "ADD".
- `static void subtractDestinationRegisterBySourceRegister(processor *proc, char source_processor_register, char destination_processor_register)` - e.g. destination_processor_register = destination_processor_register - source_processor_register, command "SUB".
- `static void incrementInstructionPointer(processor *proc)` - incrementing instruction pointer by 1.
- `static void setInstructionPointer(processor *proc, short value)` - setting instruction pointer.
- `static void increasInstructionPointerBy(processor *proc, short value)` - increasing instruction pointer by given value.
- `static void setZeroFlag(processor *proc)` - setting zero flag to true.
- `static void unsetZeroFlag(processor *proc)` - setting zero flag to false.
- `static void setSignFlag(processor *proc)` - setting sign flag to true.
- `static void unsetSignFlag(processor *proc)` - setting sign flag to false.
- `static void addRegisterToStack(processor *proc, char register)` - adds register value to stack, command "PUS".
- `static void popFromStack(processor *proc, char register)` - pops from stack to specified register, command "POP".
- `static void clearStack(processor *proc)` - cleares all data in the stack.
- `static void bitwiseOR(processor *proc, char register_one, char register_two)` - bit sum on two registers, result will be in register_one, command "OR".
- `static void bitwiseAND(processor *proc, char register_one, char register_two)` - bit product on two registers, result will be in register_one, command "AND".
- `static void bitNEG(processor *proc, char reg)` - will negate bits in given register, command "NEG".

3.2 Scheduler methods

`namespace sop::processor::Scheduler`

- `boost::shared_ptr<sop::process::Process> getHighestPriorityProcess()` - takes the highest priority process.
- `static void addProcess(boost::shared_ptr<sop::process::Process> p, sop::processor::Scheduler *sched)` - adds process to scheduler.
- `static void removeProcess(boost::shared_ptr<sop::process::Process> p, sop::processor::Scheduler *sched)` - removes process from scheduler.
- `uint8_t getUserPriority(boost::shared_ptr<sop::process::Process> p)` - from User layer, nice parameter.
- `void calculatePriority()` - calculates priorities for all tasks in both vectors.
- `void addToUnactiveTaskArray(boost::shared_ptr<sop::process::Process> p)` - adds process to unactive task array.
- `void addToActiveTaskArray(boost::shared_ptr<sop::process::Process> p)` - adds process to active task array.

- `void eraChange()` - era change, changes unactive task array to active, and active to unactive.
- `bool isEraChangeNeeded()` - checks if era change has to be done.
- `bool firstIsActive()` - checks if the first array is the active one.
- `bool secondIsActive()` - checks if the second array is the active one.
- `void clearTaskArray()` cleans vector after eraChange.
- `void printOutActiveTasks()` - prints out active tasks.
- `void printOutUnactiveTasks()` - prints out unactive tasks.
- `void printOutHelperMethod(int i, bool which)` - printing helper method.

3.3 InterpreterHandler methods

`namespace sop::processor::InterpreterHandler`

- `char getByteFromMemory(boost::shared_ptr<sop::process::Process> p)` - asks for 1 byte from memory.
- `void buildProgramLine(boost::shared_ptr<sop::process::Process> p)` - builds one program line, e.g. "ADD, A,B".
- `std::string interpretLine(boost::shared_ptr<sop::process::Process> p)` - interprets one program line, and executes it.
- `void pickDataPart(std::string s)` - gets data part from program line.
- `void pickCommandPart(std::string s)` - gets command part from program line.
- `void printDataPart()` - prints data part.
- `void printCommandPart()` - prints command part.
- `void interpreterReset()` - resets interpreter fields.
- `std::string getDataPart()` - getter for data part.
- `std::string getCommandPart()` - getter for command part.

3.4 ExecutiveUnit methods

- `sop::processor::ExecutiveUnit`
- `short getQuantTimeLeft()` - gets actual quant time that is left.
- `void resetQuantTime()` - resets quant time to standard quant time.
- `std::string processorTick()` - processor ticks once every quant time.
- `void activateProcessor()` - initial start of executive unit work.
- `void mainExecutiveLoop()` - that method manages the data that is in Scheduler and Interpreter, handles era changes, executing commands on processor, etc.

  Other in ExecutiveUnit which I'll describe in shell section - because they provide step by step operations.

4 Shell - step by step operations

4.1 Shell commands

`Shell commands are part of ExecutiveUnit class`

- `void cH_addProcess(const std::vector<const std::string> & params)` - shell command that adds process, to execute command in shell type "addProcess".
- `void cH_showQuantTimeLeft (const std::vector<const std::string> & params)` - shell command shows the quant time that is left, to execute command in shell type "quanttime".
- `void cH_showActiveTaskQueue (const std::vector<const std::string> & params)` - shell command that shows active task queue, to execute command in shell type "sacttask".
- `void cH_showUnactiveTaskQueue (const std::vector<const std::string> & params)` - shell command that shows unactive task queue, to execute command in shell type "suacttask.
- `void cH_fullTick(const std::vector<const std::string> & params)` - shell command, that makes processor do "full tick", it means that processor is executing until quant time is not zero, to execute command in shell type "fulltick".
- `void cH_showActualProcessorState(const std::vector<const std::string> & params)` - shell command, that shows actual processor state, it prints all fields of the processor, to execute command in shell type "procstate".

More about shell is written in Shell section that is Tomek's work. He implemented whole shell.

5 Interface

My main goal was to make my code as flexible as possible. That's why I figured out that good way of providing easy access to processor methods would be adding processor structure to every process. Doing that, process module was able to use my "static" ProcessorHandler class, which provides every function it needs (functions are included in "Usable methods" section). Beside this class, there're just few functions that use data from other modules. One of them is "getByteFromMemory(..)" which was asking for one byte from memory module (can be found above). Other one is "getUserPriority(..)" (can be also found above), which was checking the priority of the user from the user's module. So main communication is between Processor and Processes, and that is all included in "ProcessorHandler" class, which processes module could use at any time, the class has every function that possibly would be used.

Chapter 3

Processes

by Michał Pietrzak

---

1 Data Structures

1.1 Process structure

```
namespace sop
{
  namespace process
  {
    class Process
    {
    public:
      //__constructor__
      Process();
      //__destructor__
      ~Process();
      //__getters from protected__
      uint16_t getPID();
      uint16_t getUID();
      uint16_t getGID();
      uint16_t getArrayPages();
      uint16_t getRejestrA();
      uint16_t getRejestrB();
      uint16_t getRejestrC();
      uint16_t getRejestrD();
      //__getters from private__
      uint16_t getPPID();
      uint16_t getMemoryFlagStatus();
      uint16_t getProcessorFlagStatus();
      uint16_t getEndingFlagStatus();
      uint16_t getProcessIsInScheduler();
      uint16_t getIsActuallyRunning();
      uint16_t getIsTrueProcess();
      uint16_t getIsKilled();
      uint8_t getExitCode();
```

```
    //__setters from protected__
    void setPID(uint16_t);
    void setUID(uint16_t);
    void setGID(uint16_t);
    void setArrayPages(uint16_t);
    void setRejestrA(uint16_t);
    void setRejestrB(uint16_t);
    void setRejestrC(uint16_t);
    void setRejestrD(uint16_t);
    //__setters from private__
    void setPPID(uint16_t);
    void setMemoryFlagStatus(uint16_t);
    void setProcessorFlagStatus(uint16_t);
    void setEndingFlagStatus(uint16_t);
    void setProcessIsInScheduler(uint16_t);
    void setIsActuallyRunning(uint16_t);
    void setIsTrueProcess(uint16_t);
    void setIsKilled(uint16_t);
    void setExitCode(uint8_t);
  protected:
    //__variables__
    uint16_t _PID, _UID, _GID; //Process, User, Group Identification Number
    uint16_t _rejestrA, _rejestrB, _rejestrC, _rejestrD; //Registers
    int8_t _exitCode; //variable for exit()
    uint16_t /*sop::memory::LogicalMemory*/ _array_pages; //Template type just /*for tests*/

  private:
    //__variables__
    uint16_t _PPID;
    uint16_t _memoryFlagStatus;     // flag adjusted on 1 if memory will be allocatedc
    uint16_t _processorFlagStatus;  // flag adjusted on 1 if processor will be assigned
    uint16_t _endingFlagStatus;     // flag adjusted on 1 if proces has ended his executing
    uint16_t _processIsInScheduler; // flag adjusted on 1 if process is in scheduler
    uint16_t _isActuallyRunning;    // flag adjusted on 1 if process is Actually Running
    uint16_t _isTrueProcess;        // flag adjusted on 1 if process is 'real'
    uint16_t _isKilled;             // flag adjusted on 1 if process was killed

  };
 }
}
```

2 Code to perform the algorithm

2.1 Processes methods and other ingredients

`namespace sop::processes::Module`

- `std::vector <boost::shared_ptr<sop::process::Process>> ProcessVector` - definition of vector with process data.
- `void addToVector(boost::shared_ptr<sop::process::Process> object);` - Adds process to vector.
- `void removeFromVector(uint16_t PID);` - Deletes process with specific PID from vector.
- `boost::shared_ptr<sop::process::Process> findProcess(uint16_t PID);` - Finds process in vector of

processes (with specific PID).

- `void CreateShellInit();` - Creates INIT process which is Shell process.
- `boost::shared_ptr<sop::process::Process> createNewProcess();` - Creates new process.
- `void fork(boost::shared_ptr<sop::process::Process>);` - Creates new child of process.
- `void exec(std::string, boost::shared_ptr<sop::process::Process>);` - Writes code of program to memory.
- `void wait(boost::shared_ptr<sop::process::Process>, boost::shared_ptr<sop::process::Process>);` - Commands Parent to wait on his child.
- `void kill(boost::shared_ptr<sop::process::Process>);` - Kills specific process.
- `void exit(boost::shared_ptr<sop::process::Process>, uint8_t ExitCode);` - Exits in normal way from specific process.
- `std::queue <uint16_t> PIDlist;` - Definition of PID's queue.
- `void fillQueue();` - Fills queue with numbers from 1-100.
- `uint16_t getPIDfromList();` - Gets number from queue to new created process.

`namespace sop::process::Process`

- `Process()` - Constructor for Class Process.
- `~Process()` - Destructor for Class Process.
- `uint16_t getPID()` - Getter for PID (Process Identification).
- `uint16_t getUID()` - Getter for UID (User Identification).
- `uint16_t getGID()` - Getter for GID (Group Identification).
- `uint16_t getArrayPages()` - Getter for place where is array of pages.
- `uint16_t getRejestrA()` - Getter for Register A.
- `uint16_t getRejestrB()` - Getter for Register B.
- `uint16_t getRejestrC()` - Getter for Register C.
- `uint16_t getRejestrD()` - Getter for Register D.
- `uint16_t getPPID()` - Getter for Parent Process Identification.
- `uint16_t getMemoryFlagStatus()` - Getter for flag which shows when memory is allocated.
- `uint16_t getProcessorFlagStatus()` - Getter for flag which shows when process got Processor.
- `uint16_t getEndingFlagStatus()` - Getter for flag which shows when process has ended.
- `uint16_t getProcessIsInScheduler()` - Getter for flag which shows when process is in scheduler.
- `uint16_t getIsActuallyRunning()` - Getter for flag which shows when process is currently running.
- `uint16_t getIsTrueProcess()` - Getter for flag which shows when process is real or not.
- `uint16_t getIsKilled()` - Getter for flag which shows when process was killed or not.
- `uint8_t getExitCode()` - Getter for variable with code of exit.
- `void setPID(uint16_t)` - Setter for PID (Process Identification).
- `void setUID(uint16_t)` - Setter for UID (User Identification).
- `void setGID(uint16_t)` - Setter for GID (Group Identification)..
- `void setArrayPages(uint16_t)` - Setter for place where is array of pages.
- `void setRejestrA(uint16_t)` - Setter for Register A.
- `void setRejestrB(uint16_t)` - Setter for Register B.

- `void setRejestrC(uint16_t)` - Setter for Register C.
- `void setRejestrD(uint16_t)` - Setter for Register D.
- `void setPPID(uint16_t)` - Setter for Parent Process Identification.
- `void setMemoryFlagStatus(uint16_t)` - Setter for flag which shows when memory is allocated.
- `void setProcessorFlagStatus(uint16_t)` - Setter for flag which shows when process got Processor.
- `void setEndingFlagStatus(uint16_t)` - Setter for flag which shows when process has ended.
- `void setProcessIsInScheduler(uint16_t)` - Setter for flag which shows when process is in scheduler.
- `void setIsActuallyRunning(uint16_t)` - Setter for flag which shows when process is currently running.
- `void setIsTrueProcess(uint16_t)` - Setter for flag which shows when process is real or not.
- `void setIsKilled(uint16_t)` - Setter for flag which shows when process was killed or not.
- `void setExitCode(uint8_t)` - Setter for variable with code of exit.

3 Interface to present data changes

3.1 Commands for Shell

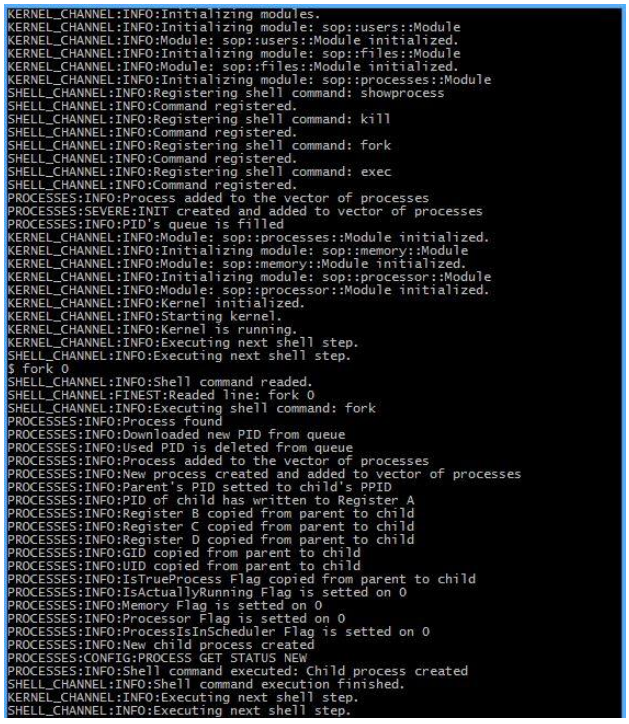
- `void cH_showprocess(const std::vector<const std::string> & params)` - Shell Command to show information about process.
- `void cH_kill(const std::vector<const std::string> & params)` - Shell Command to kill by user the process with specific PID.
- `void cH_fork(const std::vector<const std::string> & params)` - Shell Command to create a child for the process with specific PID.
- `void cH_exec(const std::vector<const std::string> & params)` - Shell Command to write a program with specific name to the process with specific PID.

4 Step by Step

4.1 Shell

To show how module is working step by step I used messages from Logger which was created by Tomasz Lewandowski. Every method have lines of code which are printing messages when something happened or something has been changed.

For example:



Messages with affix "PROCESSES::INFO" belongs to my module. Firstly we can see that INIT process is created and queue of PIDs is filled. After that we have messages from Kernel channel and after using by me command FORK are displayed more messages from my module. Text of these messages describes precisely what this command did step by step.

5 Interface to cooperate with other modules

Data from other modules

- Processor
  - Information about the process is in scheduler
  - Information about the process got processor
- Files
  - File with code of program
- Memory
  - Array of tables
  - Information about the process got memory
- Users
  - GID number
  - UID number

Data to other modules

- I share vector of processes with every information like status flags etc.

Chapter 4

Files

by Mateusz Szynka

---

1 Action concept

1.1 Filesystem

History The current structure of filesystem was based on earlier versions of EXT filesystem designed by Rémy Card on the basis of Unix File System (UFS). It was the first implementation that used virtual file system (VFS) capable of handling any possible data under it. The downside effect of EXT was high fragmentation rate along the usage therefore it was quickly replaced by EXT2.

Implementation To avoid conflicts in project as well as maintain flexability and ease of usage I've implemented the custom VFS-like structure. Since there is no physical drive to use I've also created a structure to store data in Windows text file. Class Filesystem determines the VFS as a collection of methods as well as shell handlers allowing certain filesystem operations including:

- file operations
  - creation/removal of file
  - reading/writing file data
  - searching for the file
- directory operations
  - creation/removal of directory
  - listing directory children
- statistics
  - informatut fion aboree blocks
  - graphical interpretation of disk blocks
- usage informations
  - currently opened directory, opened files
  - used blocks

Simulation Due to the fact that whole operating system is simulated on Windows machine the filesystem data structure is saved in a text file in mostly human readable form. That created an issue of class serialization which I've bypassed by some sort of pseudo-serialization module converting `class`es to `std::string` and back to `class`es.

Virtualization classes Data blocks that are the virutal equivalent of disk block are the base type for inheritation by Inode and Data blocks. Data blocks simulates RAW data, Inode `class` is linux-like inode.

2 Data structures

2.1 Filesystem

```
namespace sop
{
  namespace files
  {
    struct CurrentDirectory
    {
      std::vector<uint32_t> blockRoute;
      std::vector<std::string> path;
    };

    class Filesystem
    {
    private:
      CurrentDirectory currentDir;                                  // Current directory holder
      std::list<File*> openedFilesList;                             // List of opened files holder
      std::vector<uint32_t> freeSpace;                              // Vector of free spaces on disk
      std::array<Block*, sop::files::ConstEV::numOfBlocks> dataBlocks; // Actual disk blocks
      sop::logger::Logger* logger;                                  // Logger holder
    };
  }
}
```

2.2 File

```
namespace sop
{
  namespace files
  {
    class File
    {
      boost::shared_ptr<sop::process::Process> PIDHolder; // Holder of the opening process
          identification
      uint32_t parentCatalogAddress;          // Parent directory address holder
      uint32_t blockAddress;                  // This inode block address holder
      std::vector<std::array<char, sop::files::ConstEV::blockSize>> data; // Data holder
      char openMode;                          // Actual mode in which the file is opened (rwx)
      std::string fileName;                   // Filename holder
      uint32_t UID;                           // UID holder
      uint32_t GID;                           // GID holder
```

```
15      sop::logger::Logger* logger;            // Logger holder
    };
17  }
}
```

2.3 Block

```
namespace sop
{
  namespace files
  {
    class Block
    {
    public:
      // virtual methods
    };
  }
}
```

2.4 Inode

```
namespace sop
{
  namespace files
  {
    struct dir_u // directory helper structure
    {
      std::map<std::string,uint32_t> inodesInside;
    };
    struct file_u // file helper structure
    {
      std::array<uint32_t, sop::files::ConstEV::directAddrBlock> directBlockAddr;
      std::vector<uint32_t> indirectBlockAddr;
      uint32_t size; // dynamic size
    };
    class Inode : public Block
    {
    public:
      uint32_t uid;
      uint32_t gid;
    private:
      bool isDirectory; // directory
      dir_u directory; // directory structure
      file_u file;     // file structure
      sop::users::Permissions permissions; // users permissions
      bool lock;       // lock for one read/write operation per time
      sop::logger::Logger* logger;
    };
  }
}
```

2.5 Data

```
namespace sop
{
  namespace files
  {
    class Data : public Block
    {
      std::array<char, sop::files::ConstEV::blockSize> containter;
    };
  }
}
```

3 Usable methods

3.1 Filesystem

namespace sop::files::Filesystem

- void format() - is initilized while reading corrupted disk or as a shell command. Creates empty filesystem ready to use. Deletes all data and performs root directory tree block allocation.
- File* openFile(boost::shared_ptr<sop::process::Process> PID, std::vector<std::string> path std::string openMode) - returns pointer to a File object opened as stated in openMode parameter. PID is temporary and meant to be automatically forwarded in the end-user product. Manual due to the project linking issue.
- std::string readFile(File* fileHandler) - returns data inside opened file. The fileHandler parameter should be given from Filesystem::openFile method.
- void writeToFileP(File* file_ptr, std::string input) - saves data to given file_ptr's Inode.
- void createFile(boost::shared_ptr<sop::process::Process> PID, std::vector<std::string> path) - creates new file in given directory. The directory path from the std::string might be traced by std::vector<std::string> getPathFromParam(std::string path); function in filesystem.h.
- void saveFile(File* fileHandler) - stores data on hard drive. Equal to any file save operation in any operating system.
- void closeFile(File* fileHandler) - closes the file by erasing it from openedFiles vector and changes inode lock flag to 0.
- void removeFile(boost::shared_ptr<sop::process::Process> PID, std::vector<std::string> path) - removes file's inodes and data blocks. That will cause severe data loss. Not possible if path don't exist or no permission.
- void writeToFile(File* fileHandler, std::string data); - stores data in file's data blocks given in std::string data. File must be opened in write mode and has to have permissions to write.
- File* seek(boost::shared_ptr<sop::process::Process> PID, std::vector<std::string> path) - searches for the file in path given in std::vector<std::string> path. Returns pointer to searched File or nullptr if not found. To translate path from string use std::vector<std::string> getPathFromParam(std::string path) function from filesystem.h library.
- std::string getCurrentDir() - returns currently opened directory for shell usage as a std::string.
- std::string getCurrentPath() - returns path to currently opened directory for shell usage as a std::string.

- void changeDirectory(boost::shared_ptr<sop::process::Process> PID, std::vector<std::string> path) - modifies currently opened directory to given path. If path doesn't exist there is undefined behaviour.
- void changeDirectoryUp() - shortcut modifying directory to last element's parent. If path doesn't exist there is undefined behaviour.
- void createDirectory(boost::shared_ptr<sop::process::Process> PID, std::vector<std::string> path) - creates directory in path given. If parent of path doesn't exist directory will not be created.
- void removeDirectory(boost::shared_ptr<sop::process::Process> PID, std::vector<std::string> path) - removes directory given in path. If directory doesn't exist there is undefined behaviour.
- std::vector<dirList> list() - returns std::vector of structures dirList. Helper structure is given in namespace sop::files as follows:

```
struct dirList
{
  std::string drwx;
  std::string size;
  std::string name;
  uint32_t block;
  uint32_t username;
  uint32_t group;
};
```

- void printStats() - prints disk statistics on std::cout. Prints freeSpace vector, openedFiles list and currentPath vector.
- void printDisk(uint32_t parts) - prints disk blocks on std::cout parted as given in parts parameter.
- void printDiskTree(uint32_t depth) - prints disk directory tree on std::cout to the level given in depth. The algorithm uses pre-order tree traversal method.
- void printDataBlock(uint32_t block) - prints statistics on data block as given in block parameter on std::cout. For directory prints: "This is a directory!", for file prints data from given block.
- void changeDirectoryHandler(const std::vector<const std::string> & params) - shell handler for change directory method. Reads path from params.
- void removeFileHandler(const std::vector<const std::string> & params) - shell handler for file removal.
- void viHandler(const std::vector<const std::string> & params) - shell handler for running vi-styled file editor. Reads path from params. If file doesn't exist there is undefined behavior.
- void createFileHandler(const std::vector<const std::string> & params) - shell handler for file creation. Runs sop::files::Filesystem::createFile() with given path.
- void createDirectoryHandler(const std::vector<const std::string> & params) - shell handler for directory creation. Runs sop::files::Filesystem::createDirectory() with given path.
- void removeDirectoryHandler(const std::vector<const std::string> & params) - shell handler for directory removal. Runs sop::files::Filesystem::removeDirectory() with given path.
- void catHandler(const std::vector<const std::string> & params) - shell handler. Prints data from file given in params on std::cout. If file not found the information will be printed on std::cout.
- void listHandler(const std::vector<const std::string> & params) - shell handler for listing directory.

- `void statHandler(const std::vector<const std::string> & params)` - shell handler for printing filesystem statistics. Uses function `sop::files::Filesystem::printStats()` and `sop::files::Filesystem::printDisk(16)`.
- `void formatHandler(const std::vector<const std::string> & params)` - shell handler for formating filesystem to blank. Could be used only with root user account.

3.2 File

`namespace sop::files::File`

- `uint32_t getBlockAddr()` - returns number of block where the administrative inode block is written on disk
- `std::vector<std::array<char, sop::files::ConstEV::blockSize>> getData()` - returns RAW data stored in file as a `std::vector`. If thoere is no data in file the returned vector size will be 0.
- `char getMode()` - return mode in which file is opened. Standard Linux-styled rwx.
- `std::string getFileName()` - return filename of opened File. Could be used to extract path using filesystem's `sop::files::Filesystem::seek()`.
- `uint32_t getUID()` - returns user ID. Required for authorization by users module.
- `uint32_t getGID()` - returns group ID. Required for authorization by users module.
- `void writeToFile(std::string, std::vector<uint32_t>* freeSpace)` - writes data to currently opened file as given in `std::string` parameter.
- `void sop::files::File::removeFile(std::vector<uint32_t>* freeSpace)` - removes currently opened file. Should be used only from filesystem layer.
- `void setMode(char mode)` - sets mode in which the file will be opened. Required for authorization by users module.
- `Inode* getInode()` - returns Inode object of currently opened file. Required for authorization by users module.
- `uint32_t getSize()` - return size of data in currently opened file. If it is a directory it returns 0.
- `void setFilename(std::string)` - renames currently opened file.
- `boost::shared_ptr<sop::process::Process> getPID()` - returns pointer to process that opened this file.
- `sop::users::Permissions getPermission()` - returns `sop::users::Permissions` object defined in `sop::users`. Required for authorization by users module.

3.3 Serialization module

`namespace sop::files::Serialize`

- `void save()` - creates a `std::string` with all the data inside Filesystem object and then using `std::fstream` pushes it to the file - VFS.
- `void read()` - reads the file with the filename given in `filename` holder `std::string` and converts it line by line to objects.

Chapter 5

Users

by Tomasz Lewandowski

---

1 General Overview

1.1 Description

The Users Module contains tools to manage users, groups, permissions and priorities. All the classes were designed in a way that allows easy access to the utilities for the other team members. The interface for other modules is flexible, many methods are overloaded so they can work for example either on uid or username. For these that does not have counterparts there is always a simple way around by using methods that finds the user/group in the system, so when having username it is easy to get it's user id in one or two steps. The way the module was designed allows it to be really easy extended, for example by providing new Encryption classes or modifying the values listed in this section.

1.2 Type definitions

The essential types used in the module. It is very important to define them in one place and use everywhere instead of types like "int" to avoid bugs when refactoring. There is always a possibility that something will be changed. These type definitions allows to do changes in one place instead of searching through all the files in the project.

```
namespace sop
{
  namespace users
  {
    typedef uint16_t uid_t;
    typedef uid_t gid_t;
    typedef int8_t priority_t;
    typedef uint8_t permission_t;
  }
}
```

1.3 Built-in constants

Declarations

```
namespace sop
{
  namespace users
  {
    extern const uid_t kMax_system_uid; //max uid for system purposes
    extern const gid_t kMax_system_gid; //max gid for system purposes
    extern const uid_t kMax_uid; //max uid
    extern const gid_t kMax_gid; //max gid
    extern const priority_t kMin_priority; //beginning of the priorities interval
    extern const priority_t kDefault_priority; //default group priority
    extern const priority_t kMax_priority; //end of the priorities interval
  }
}
```

Default values

```
const sop::users::uid_t sop::users::kMax_system_uid=999;
const sop::users::gid_t sop::users::kMax_system_gid=999;
const sop::users::uid_t sop::users::kMax_uid=32768;
const sop::users::gid_t sop::users::kMax_gid=32768;
const sop::users::priority_t sop::users::kMin_priority=-5;
const sop::users::priority_t sop::users::kDefault_priority=0;
const sop::users::priority_t sop::users::kMax_priority=4;
```

2 Data structures

2.1 User

The User structure represents and stores information about a single user. Creating the user by constructor does not mean that it is in the system and can be logged on. To do so, it has to be added to the User Manager. Nobody is a fictional user that holds the maximum uid and gid that can be used in the system.

```
namespace sop
{
  namespace users
  {
    struct User{
      User();
      User(uid_t uid, gid_t gid, const std::string & username, const std::string & password, const
          std::string & info, const std::string & home_dir);
      uid_t uid;
      gid_t gid;
      std::string username;
      std::string password; //encrypted password
      std::string info;
      std::string home_dir;
    };
```

```
15
    extern boost::shared_ptr<User> nobody;
17  }
}
```

2.2 Group

The Group structure represents and stores information about a single group. Creating the group by constructor does not mean that it is in the system. To do so, it has to be added to the Group Manager. Nogroup is a fictional group that holds the maximum gid that can be used in the system.

```
namespace sop
2 {
  namespace users
4   {
    struct Group
6     {
      Group();
8     Group(gid_t gid, const std::string & group_name);
      gid_t gid;
10    std::string group_name;
      std::list<boost::shared_ptr<User>> users_list;
12    };

14    extern boost::shared_ptr<Group> nogroup;
  }
16 }
```

2.3 Permissions

The Permissions structure represents file's permissions for user, group and the others.

```
namespace sop
2 {
  namespace users
4   {
    struct Permissions
6     {
      static const permission_t kRWX=7;
8     static const permission_t kRW=6;
      static const permission_t kRX=5;
10    static const permission_t kR=4;
      static const permission_t kWX=3;
12    static const permission_t kW=2;
      static const permission_t kX=1;
14    static const permission_t kNone=0;

16    Permissions(); //000
      Permissions(bool for_directory); //dir 777 file 666
18    Permissions(permission_t user, permission_t group, permission_t others);

20    permission_t user;
```

```
      permission_t group;
      permission_t others;
    };
  }
}
```

3 Classes

Due to the fact that classes are large in size, only the fields, constructors and destructors of these classes are presented below. To see the most important methods of the following classes go to the **Usable methods** section.

3.1 Module

The Module class provides access to all components of the Users Module so the other modules can use them. It also groups all handlers for shell commands.

```
namespace sop
{
  namespace users
  {
    class Module : public sop::system::Module
    {
      public:
        explicit Module(sop::system::Kernel *kernel);
        virtual ~Module();

      protected:

      private:
        UsersManager _users_manager;
        GroupsManager _groups_manager;
        PriorityManager _priority_manager;
        PermissionsManager _permissions_manager;
    };
  }
}
```

3.2 Users Manager

Users Manager class keeps the list of all the users in the system and provides utilities that can be used to manage them.

```
namespace sop
{
  namespace users
  {
    class UsersManager : public Object
    {
      public:
        static const boost::regex username_regex;
        static const boost::regex password_regex;
```

```
10
        explicit UsersManager(Module *module);
12      virtual ~UsersManager();
        virtual std::string getClassName() const;
14
      protected:
16      Module *_module;
        boost::shared_ptr<Encryptor> _encryptor;
18      std::list<boost::shared_ptr<User>> _users_list;

20    private:

22    };
  }
24 }
```

The username regex and password regex are defined as follows.

```
const boost::regex sop::users::UsersManager::username_regex =
    boost::regex("^[a-zA-Z][0-9a-zA-Z_]*$");
2 const boost::regex sop::users::UsersManager::password_regex =
    boost::regex("^(([a-zA-Z])([a-zA-Z0-9!@#$%^&*_]){2,}))?$");
```

3.3 Groups Manager

The Group Manager class keeps the list of all the groups in the system and provides utilities that can be used to manage them.

```
namespace sop
2 {
  namespace users
4 {
    class GroupsManager : public Object
6   {
      public:
8       static const boost::regex group_name_regex;

10      explicit GroupsManager(Module *module);
        virtual ~GroupsManager();
12
      protected:
14      Module *_module;
        std::list<boost::shared_ptr<Group>> _groups_list;
16
      private:
18
    };
20 }
}
```

The groupname regex is defined as follows.

```
const boost::regex sop::users::GroupsManager::group_name_regex =
    boost::regex("^[a-zA-Z][0-9a-zA-Z_]*$");
```

3.4 Permissions Utilities

The class allows to convert permissions to RWX strings and vice versa.

```
namespace sop
{
  namespace users
  {
    class PermissionsUtilities
    {
      public:
      protected:
      private:
        PermissionsUtilities();
    };
  }
}
```

3.5 Permissions Manager

Permissions Manager class allows to manage files permissions as well as checking if the user has permissions to that file.

```
  namespace users
  {
    class PermissionsManager : public Object
    {
      public:
        explicit PermissionsManager(Module *module);
        virtual ~PermissionsManager();

      protected:
        Module *_module;

      private:

    };
  }
}
```

3.6 Priority Manager

This class manages the priorities of individual groups.

```
namespace sop
{
  namespace users
```

```
4   {
      class PriorityManager : public Object
6     {
        public:
8         explicit PriorityManager(Module *module);
          virtual ~PriorityManager();
10      protected:
          Module *_module;
12        std::map<gid_t,priority_t> priorities;
        private:
14
      };
16  }
}
```

3.7 Encryptors

Encryptor is an abstract class that represents any encryption technique or hash function. It provides one direction encryption method that can be used to encrypt passwords. The only implementation of the Encryptor is CaesarEnryptor that uses Caesar's cripher algorithm.

```
1  namespace sop
{
3   namespace users
    {
5     class Encryptor
      {
7       public:
          Encryptor();
9         virtual ~Encryptor();
        protected:
11        virtual void doEncrypt(const std::string & data, std::string * encrypted_data) const = 0;
        private:
13    };

15    class CaesarEncryptor : public Encryptor
      {
17      public:
          CaesarEncryptor(uint8_t shift);
19        ~CaesarEncryptor();
        protected:
21        uint8_t _shift;
          virtual void doEncrypt(const std::string & data, std::string * encrypted_data) const;
23      private:
          CaesarEncryptor();
25    };
    }
27  }
```

4 Usable methods

This section contains the most important methods of the classes included in the module. Every class that has an access to the Module or the Kernel uses Logger to inform about everything that happens in the system during the execution of the methods. Many methods take constant references as parameters to avoid unnecessary copying of the data.

4.1 Module

The only usable methods of modules is the family of `X* getXManager();` methods where X is the name of the manager. Every method in that family returns the pointer to the manager that it represents. They provide access to the managers for other modules and managers. The list of the methods in the family:

- `UsersManager* getUsersManager();`
- `GroupsManager* getGroupsManager();`
- `PriorityManager* getPriorityManager();`
- `PermissionsManager* getPermissionsManager();`

4.2 Users Manager

- `bool addUser(const User & user);` - creates dynamic copy of the given user and adds it to the list of the users in the system. Returns true on success or false otherwise (e.g. when username is already taken).
- `bool deleteUser(const std::string & username);` - deletes user with the given username from the system. Returns true on success or false otherwise.
- `boost::shared_ptr<User> findUser(uid_t uid);` - returns the shared pointer to the user specified by the uid if it exists or null pointer otherwise.
- `boost::shared_ptr<User> findUser(const std::string & username);` - returns the shared pointer to the user specified by the username if it exists or null pointer otherwise.
- `uid_t getNextFreeUID(uid_t greater_than = kMax_system_uid);` - calculates and returns the first user id that is free and greater than *greater_than*. If there is no free user id then the uid of nobody user is returned.
- `bool isUIDFree(uid_t uid);` - returns true if the given uid is free or false otherwise
- `bool isUsernameFree(const std::string & username);` - returns true if the given username is free or false otherwise.
- `bool login(boost::shared_ptr<sop::process::Process> process, const std::string & username, const std::string & password);` - logins user on the given process if the username and password are correct. On success it changes the uid of the process and returns true. On failure it returns false and makes no changes in the process.

4.3 Groups Manager

- `bool addGroup(const Group & group);` - creates dynamic copy of the given group and adds it to the list of the groups in the system. Returns true on success or false otherwise (e.g. when groupname is already taken).
- `bool deleteGroup(const std::string & group_name);` - deletes group with the given group name from the system. Returns true on success or false otherwise (e.g. group is not empty so it cannot be deleted).

- `boost::shared_ptr<Group> findGroup(gid_t gid);` - returns the shared pointer to the group specified by the gid if it exists or null pointer otherwise.
- `boost::shared_ptr<Group> findGroup(const std::string & group_name);` - returns the shared pointer to the group specified by the group name if it exists or null pointer otherwise.
- `gid_t getNextFreeGID(gid_t greater_than = kMax_system_gid);` - calculates and returns the first group id that is free and greater than *greater_than*. If there is no free group id then the gid of nogroup is returned.
- `bool isGIDFree(gid_t gid);` - returns true if the given gid is free or false otherwise
- `bool isGroupNameFree(const std::string & group_name);` - returns true if the given group name is free or false otherwise.

4.4 PermissionsManager

- `bool hasPermission(sop::files::Inode *node, boost::shared_ptr<sop::process::Process> process, permission_t mode);` - returns true, if the user logged on the process has permission to open a file represented by the inode in the given mode, or false otherwise.
- `bool changePermissions(sop::files::Inode *node, boost::shared_ptr<sop::process::Process> process, Permissions permissions);` - change the permissions of the file represented by inode if the user logged on the process has write permissions to that file. Returns true on success or false otherwise.
- `bool changeOwner(sop::files::Inode *node, boost::shared_ptr<sop::process::Process> process, uid_t new_uid);` - changes the owner of the file to the given user id. Operation requires that the user logged on the process is a superuser. Returns true on success or false otherwise.
- `bool changeGroup(sop::files::Inode *node, boost::shared_ptr<sop::process::Process> process, uid_t new_gid);` - changes the group of the file to the given group id. Operation requires the user logged on the process to be in the target group and has write permissions to the file. Return true on success or false otherwise
- `bool isSuperUser(boost::shared_ptr<sop::process::Process> process);` - returns true if user logged on the process is a superuser (uid==0) or false otherwise.

4.5 PriorityManager

- `priority_t getUserPriority(uid_t uid);` - returns the priority of the user specified by the uid. In fact it returns the priority of the group that user belongs to.
- `priority_t getGroupPriority(gid_t gid);` - returns the priority of the group specified by the gid. If the priority of the group is not specified in the system then the default priority is returned.
- `void setUserPriority(uid_t uid, priority_t priority);` - sets the priority of of the user specified by uid to the priority specified by priority. In fact it sets the priority of the group that user belongs to.
- `void setGroupPriority(gid_t gid, priority_t priority);` - sets the priority of the group specified by gid to the priority specified by priority.
- `void removeGroupPriorityEntry(gid_t gid);` - removes the priority of the group specified by the gid. From this point the group will have default priority until set again.

4.6 Encryptors

- `std::string encrypt(const std::string & data)const;` - invokes doEncrypt method on the given data and returns the string with encrypted data as a result.

- `virtual void doEncrypt(const std::string & data, std::string * encrypted_data)const = 0;` - virtual method that every Encryptor has to implement. Encrypts the content of the data and writes the encrypted data to the string that is in the memory at an address specified by encrypted_data.

5 Shell commannds handlers

The shell commands provides tools and utilities for the system user to interact with the module. They allows to add users, groups, show informations about them and delete them, change passwords etc. Every command is separate therefore there is possibility to see the state of the system at any given time.

5.1 Users managment commands

- `void cH_useradd(const std::vector<const std::string> & params);` - handles the shell command that allows to create new user. Creation of the user results in creating a group with the same name and gid. User can have specified uid hovewer it has to be free. If the group id is specified then this group has to exist. Execution of the command requires superuser permissions.
- `void cH_userfind(const std::vector<const std::string> & params);` - handles the shell command that finds and prints the extended information about user specified by username or uid.
- `void cH_userdel(const std::vector<const std::string> & params);` - handles the shell command that allows to delete user with the given username. Requires superuser permission.
- `void cH_userslist(const std::vector<const std::string> & params);` - handles the shell command that allows to print information about all the users in the system.
- `void cH_chpasswd(const std::vector<const std::string> & params);` - handles the shell command that allows to change user's password
- `void cH_whois(const std::vector<const std::string> & params);` - handles the shell command that prints the username of the currently logged user
- `void cH_login(const std::vector<const std::string> & params);` - handles the shell command that allows to login on the different username. If the username has a password it has to be typed as a parameter.

5.2 Groups managment commands

- `void cH_groupadd(const std::vector<const std::string> & params);` - handles the shell command that allows to create new group. Execution of the command requires superuser permissions.
- `void cH_groupfind(const std::vector<const std::string> & params);` - handles the shell command that prints information about the group specified by gid or group name.
- `void cH_groupdel(const std::vector<const std::string> & params);` - handles the shell command that deletes the group specified by group name. It requires deleting group to be empty or it will not be deleted. Execution of the command requires superuser permissions.
- `void cH_groupslist(const std::vector<const std::string> & params);` - handles the shell command that prints information about all groups in the system.
- `void cH_groupmembers(const std::vector<const std::string> & params);` - handles the shell command that prints all the members of group specified by the group name.
- `void cH_groupchange(const std::vector<const std::string> & params);` - handles the shell command that allows to change the group that the user belongs to. Execution of the command requires superuser permissions.

5.3 Permissions managment commands

- `void cH_access(const std::vector<const std::string> & params);` - handles the shell commands that allows the user to check if he has permissions to open a file in the given mode.
- `void cH_chmod(const std::vector<const std::string> & params);` - handles the shell command that allows to change permissions to a file. Execution of the command requires write permissions.
- `void cH_chown(const std::vector<const std::string> & params);` - handles the shell command that allows to change the owner of a file. Execution of the command requires superuser permissions.
- `void cH_chgrp(const std::vector<const std::string> & params);` - handles the shell commnd that allows to change the group owner of a file. Execution of the command requires that the user is in target group and has write permissions.

5.4 Priority managment commands

- `void cH_nice(const std::vector<const std::string> & params);` - handles the shell command that allows to change a nice priority of the group. Setting a priority below zero requires superuser permissions.
- `void cH_shownice(const std::vector<const std::string> & params);` - handles the shell command that prints the nice priority of the user or the group.
- `void cH_removeniceentry(const std::vector<const std::string> & params);` - handles the shell command that removes the nice priority of the group. From this point the group will have a default priority.

Project summary

Unit testing Overall almost every member of group used technique called test-driven development allowing at any stage to maintain usability, effectiveness and error-free operations. As a result the function calls, shell handlers and functionality at any point was maximized and stable.

Merge issues At the merge point of the project development our group has encountered some design assumption mismatches with the final product. What we expected from each other wasn't included in the module code and adding that with an unsuccessful timing resulted in overall design failure causing whole project to fail. Due to that kind of issues almost every module ended to be single apart from the filesystem module and users module which were at last merged together.

Final product The final operating system has never been finished. The only modules were the top-level ones taking care of security, permissions and data entries management. Although possible failure the project has given us an opportunity to learn team coding, basis of a Linux OS and better understand how independent work could break dependable machine.

"Anything that can possibly go wrong, does." - Murphy's law